

面向对象程序设计基础

(OOP)

黄民烈

aihuang@tsinghua.edu.cn

<http://coai.cs.tsinghua.edu.cn/>

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- 1.1 命令提示符
- 1.2 环境变量设置
- 1.3 主流编译器及IDE
- 1.4 ssh远程登录与操作

本讲内容提要

- 1.5 源程序的结构、编译、链接
- 1.6 多文件编译和链接过程
- 1.7 宏定义
- 1.8 编写Make工具的脚本程序
- 1.9 使用程序主函数的命令行参数
- 1.10 GDB调试工具

源程序的结构

```
#include <iostream>  
using namespace std;
```

头文件与编译指令

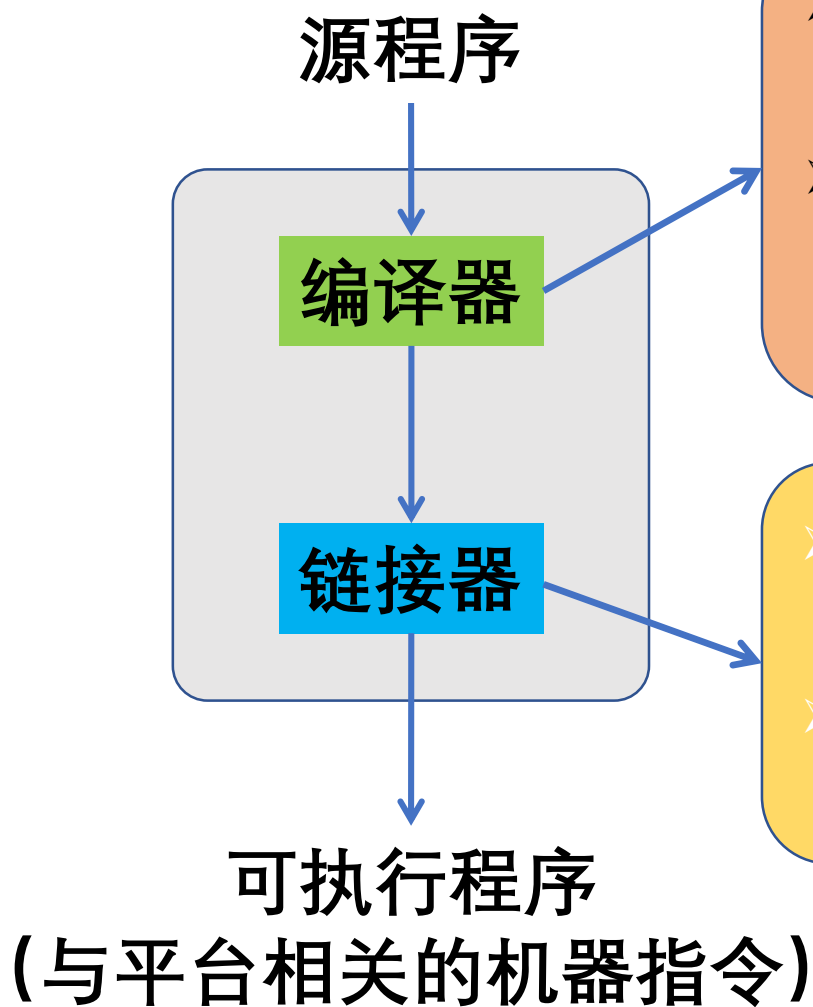
```
int add(int a, int b)  
{  
    return a + b;  
}
```

辅助函数定义

```
int main()  
{  
    cout << add(3, 4);  
    return 0;  
}
```

主函数定义

编译、链接



- 第一遍执行语法分析和静态类型检查，将源代码解析为语法分析树的结构
- 第二遍由代码生成器遍历语法分析树，把树的每个节点转换为汇编语言或机器代码，生成目标模块(.o或.obj文件)

- 把一组目标模块链接为可执行程序，使得操作系统可以执行它
- 处理目标模块中的函数或变量引用，必要时搜索库文件处理所有的引用

进一步阅读：
《C++编程思想》
2.1语言的编译过程

```
dmye@ubuntu:~$ ls ex1.*
```

```
ex1.cpp
```

```
dmye@ubuntu:~$ cat ex1.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    cout << "Hello, OOP" << endl;
```

```
    return 0;
```

```
}
```

```
dmye@ubuntu:~$ g++ -c ex1.cpp
```

```
dmye@ubuntu:~$ ls ex1.*
```

```
ex1.cpp ex1.o
```

```
dmye@ubuntu:~$ g++ -o ex1.out ex1.o
```

```
dmye@ubuntu:~$ ls ex1.*
```

```
ex1.cpp ex1.o ex1.out
```

```
dmye@ubuntu:~$ ./ex1.out
```

```
Hello OOP
```

只编译
不链接

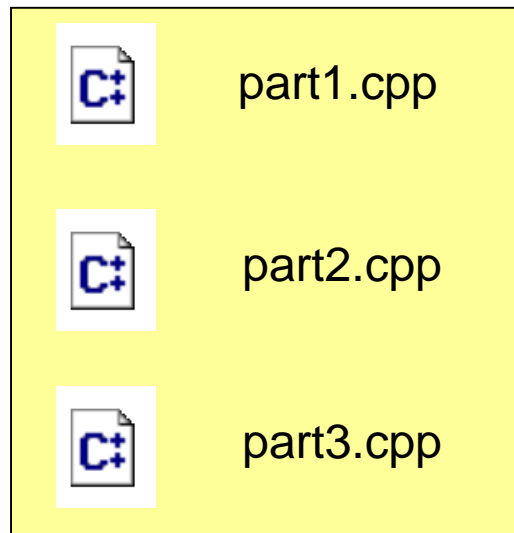
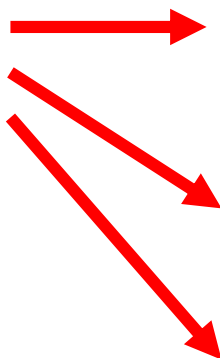
链接
程序

多个源文件的编译与链接

EX4.cpp



EX4



EX4

单个源文件的编译与链接

编译: g++ -o ex5 ex5.cpp

```
// ex5.cpp @ 20200129
```

```
#include <iostream>
```

```
#include <cstdlib> // atoi()
```

```
int ADD(int a, int b) { return a + b; }
```

```
int main(int argc, char** argv) {
```

```
    if (argc != 3) {
```

```
        std::cout << "Usage: " << argv[0]
```

```
                << " op1 op2" << std::endl;
```

```
        return 1;
```

```
    }
```

```
    int a, b;
```

```
    a = atoi(argv[1]); b = atoi(argv[2]);
```

```
    std::cout << ADD(a, b) << std::endl;
```

```
    return 0;
```

```
}
```


多个源文件的编译与链接

```
// ex5_main.cpp @ 20200129
```

```
#include <iostream>
```

```
#include <cstdlib> // atoi()
```

```
int ADD(int a, int b);
```

```
int main(int argc, char** argv) {
```

```
    if (argc != 3) {
```

```
        std::cout << "Usage: " << argv[0]
```

```
                << " op1 op2" << std::endl;
```

```
        return 1;
```

```
    }
```

```
    int a, b;
```

```
    a = atoi(argv[1]); b = atoi(argv[2]);
```

```
    std::cout << ADD(a, b) << std::endl;
```

```
    return 0;
```

```
}
```

```
// func.cpp
```

```
int ADD(int a, int b)
```

```
{ return a + b; }
```

拆解 **声明** 和 **定义**

多个源文件的编译与链接

1) 直接编译 (g++帮我们省略了一些步骤)

```
dmye@ubuntu:~$ ls
ex5_main.cpp  func.cpp  func.h
dmye@ubuntu:~$ g++ ex5_main.cpp func.cpp -o
test1
dmye@ubuntu:~$ ls
ex5_main.cpp  func.cpp  func.h  test1
dmye@ubuntu:~$ ./test1 3 4
7
dmye@ubuntu:~$ rm test1
```

删除test1

多个源文件的编译与链接

2) 分步编译（实际运行步骤）

只编译
不链接

```
dmye@ubuntu:~$ ls
ex5_main.cpp func.cpp func.h
dmye@ubuntu:~$ g++ -c ex5_main.cpp -o main.o
dmye@ubuntu:~$ g++ -c func.cpp -o func.o
dmye@ubuntu:~$ ls
ex5_main.cpp func.cpp func.h func.o main.o
dmye@ubuntu:~$ g++ main.o func.o -o test2
dmye@ubuntu:~$ ls
ex5_main.cpp func.cpp func.h func.o main.o
test2
dmye@ubuntu:~$ ./test2 3 4
```

链接

- 链接: 将各个目标文件中的各段代码进行地址定位, 生成与特定平台相关的可执行文件
- 外部函数的**声明** (一般声明在头文件中) 只是令程序顺利通过编译, 此时并不需要搜索到外部函数的**实现 (或定义)**。
- 在链接过程中, 外部函数的**实现 (或定义)** 才会被寻找和添加进程序, 一旦没有找到**函数实现**, 就无法成功链接。

链接 -- 函数和全局变量

- 为什么只在头文件(.h)定义函数声明而不实现函数体（即定义）？
 - 若函数没定义成局部函数而又有多个cpp文件包含此头文件，在链接时因发现多个相同的函数实现而发生错误
- 若头文件中定义全局变量且多个cpp文件包含此头文件，在链接时会因重复定义而发生错误

多个源文件的编译与链接

源文件

ex5_main.cpp

func.cpp

编译

编译

目标文件

main.o

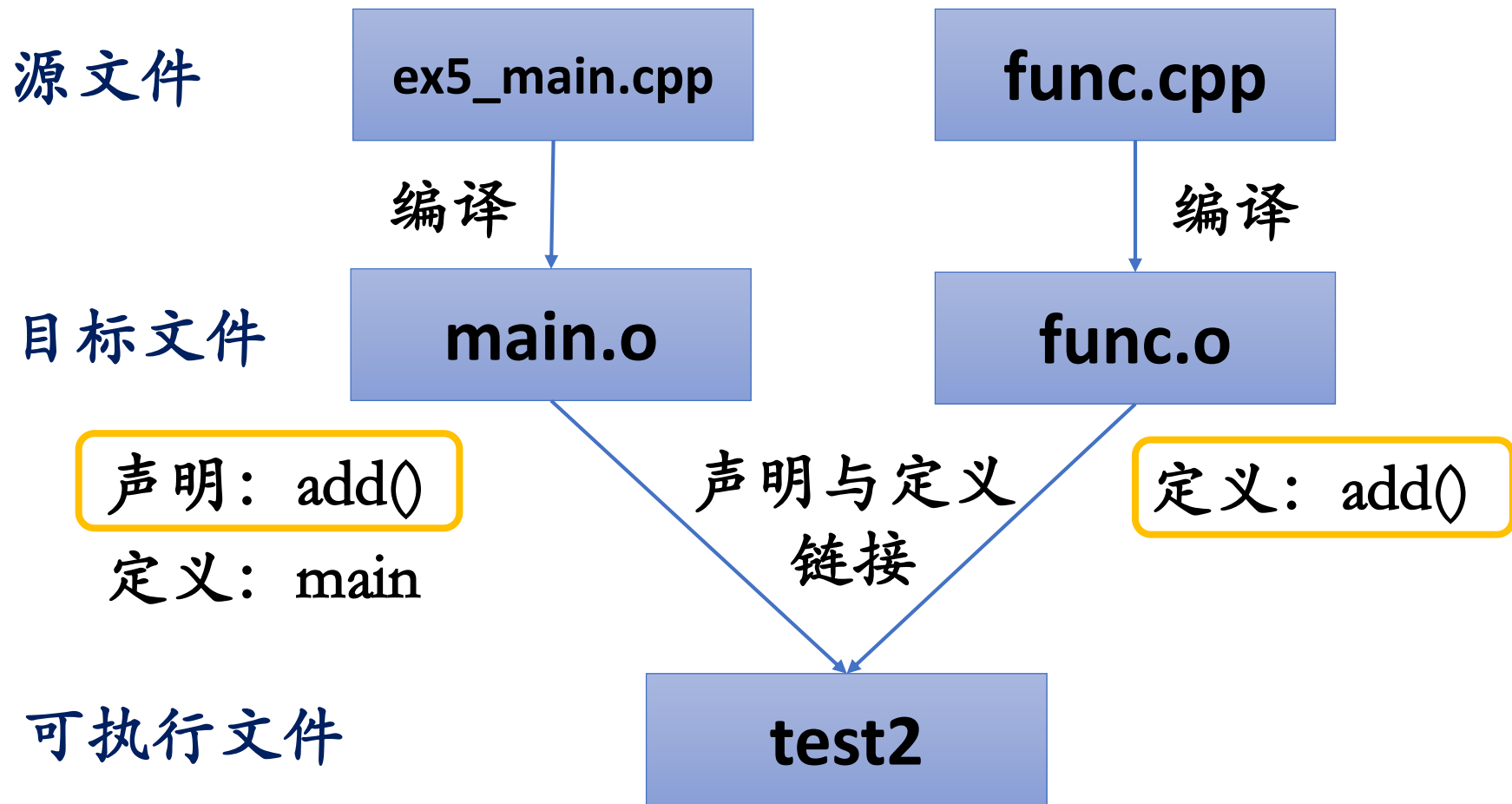
func.o

声明: add()

定义: main()

定义: add()

多个源文件的编译与链接



链接错误

```
// ex5_main.cpp @ 20200129
```

```
#include <iostream>
```

```
#include <cstdlib> // atoi()
```

```
int ADD(int a, int b);
```

```
int main(int argc, char** argv) {
```

```
    if (argc != 3) {
```

```
        std::cout << "Usage: " << argv[0]
```

```
                << " op1 op2" << std::endl;
```

```
        return 1;
```

```
    }
```

```
    int a, b;
```

```
    a = atoi(argv[1]); b = atoi(argv[2]);
```

```
    std::cout << ADD(a, b) << std::endl;
```

```
    return 0;
```

```
}
```

```
// func.cpp
```

```
float ADD(float a, float b)
```

```
{ return a + b; }
```

声明和定义不一致

链接错误

目标文件

main.o

func.o

声明: **add(int, int)**

定义: `main()`

声明与定义
链接

定义: **add(float, float)**

可执行文件

test2

编译指令: `g++ ex5_main.cpp func.cpp -o main.out`

main.o: In function `main':

main.cpp:(.text+0x8f): **undefined reference** to `ADD(int, int)'

collect2: error: ld returned 1 exit status

链接错误: 未定义的引用

使用头文件

- 有时辅助函数(如全局函数)会在多个源文件中被使用
- 头文件(.h)
 - 避免反复编写同一段声明
 - 统一辅助函数的声明, 避免错误
- #include 预编译指令
 - 将被包含的文件代码, **直接复制**到当前文件
 - 一般被用于包含头文件 (实际也能包含任意代码)

```

// ex5_main.cpp
#include <bits/stdc++.h>
using namespace std;

#include "func.h" // 定义了ADD函数

int main(int argc, char** argv)
{
    if (argc != 3)
    {
        cout << "Usage: " << argv[0]
              << " op1 op2" << endl;
        return 1;
    }

    int a, b;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    cout << ADD(a, b) << endl;
    return 0;
}

```

```

// func.h
int ADD(int a, int b);

```

```

// func.cpp
#include "func.h"
int ADD(int a, int b)
{
    return a + b;
}

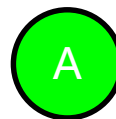
```

```
// func.h  
int ADD(int a, int b);
```

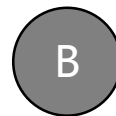
```
// func.cpp  
#include "func.h"  
int ADD(int a, int b)  
{ return a + b; }
```

```
// ex5_main.cpp  
#include <iostream>  
#include <cstdlib> // atoi()  
#include "func.h" // ADD()  
int main(int argc, char** argv) {  
    .....  
    std::cout << ADD(a, b) << std::endl;  
    return 0;  
}
```

在我们的例子中
去掉红色语句(include)是
否仍然能够成功编译及链接



可以



不可以

提交

g++编译中-c选项的意思是

- ☐ A 生成程序级调试信息
- ☐ B 定义输出名称
- ☐ C 只链接程序
- ☒ D 只编译程序而不链接程序

提交

声明与定义

- 函数声明

```
int ADD(int a, int b);
```

```
int ADD(int, int); //变量名可省略
```

- 函数定义 (也叫实现)

```
int ADD(int a, int b) {return a + b;}
```

- 同一个函数可以有多个声明，但只能有一次实现

- 多次实现会导致链接错误

```
func.o: In function `ADD(int, int)':
```

```
func.cpp:(.text+0x0): multiple definition of `ADD(int, int)'
```

```
main.o:main.cpp:(.text+0x0): first defined here
```

声明与定义

- 变量也可以有声明和定义

- 变量定义
`int x = 0; //定义并初始化`
`int arr[100]; //定义数组`

- 问题： 以下是变量声明还是变量定义？

`int x;`

声明与定义

```
// num.cpp  
int a = 1;
```

```
// ex6.cpp  
int a;  
int main() {  
    a += 1;  
    return 0;  
}
```

编译指令: `g++ ex6.cpp num.cpp -o main.out`

num.o:(.bss+0x0): **multiple definition** of `a'
main.o:(.bss+0x0): first defined here

声明与定义

- 是变量定义!

int x; //定义但不初始化

- 如何进行变量声明?

小知识：全局变量即使不初始化，也会默认为0。

extern关键字

- 变量的声明: **extern**关键字

extern int x; //声明变量

extern int arr[100]; //声明数组变量

- extern关键字也可以用于函数声明

- 但extern对于函数声明不是必须的。

extern int ADD(int a, int b);

- extern通常用在全局变量在不同文件内的共享

```
// num.h, 本组均是声明
extern int a;
extern int b;
extern int c[5];
extern int d;
```

```
// num.cpp, 本组均是定义
#include "num.h"
int a = 1;
int d = 4;
int c[5];
```

```
// ex6.cpp
#include "num.h"
int b = 2; //由于之前已经声明，这里也可以写定义
int main()
{
    a += 1;
    d += 1;
    c[0] = 1;
    cout << a << " " << b << " " << c[0] << " " << d << endl;
    return 0;
}
```

输出结果: 2 2 1 5

```
// func.cpp
int x = 0;
int add(int a) {
    x += a;
    return x;
}
```

```
// main.cpp
_____ (填空) _____
int main() {
    add(1);
    return 0;
}
```

划线处可以添加多条语句。
在保证程序能够正常编译、
链接的情况下，****至少**一
定要**添加的语句有？

- ☐ A int x;
- ☐ B extern int x;
- ☒ C int add(int);
- ☐ D int add(int x) {return 0;}

编译指令：g++ main.cpp func.cpp -o main.out

提交

宏定义的使用

#define是C++语言中的一个**预编译指令**，它用来将一个标识符定义为一个字符串，该标识符被称为**宏名**，被定义的字符串称为**替换文本**。

在程序被编译前，先将宏名用被定义的字符串替换，这称为宏替换，替换后才进行编译，宏替换是简单的替换。

宏定义的使用

1) 简单的宏替换

`#define` <宏名> <字符串>

例:

```
#define PI 3.1415926535
```

在C++中，这种替换一般被`const`取代，进而能保证类型的正确性。

例:

```
const double PI = 3.1415926535
```

宏定义的使用

2) 带参数的宏定义

#define <宏名>(<参数表>) <字符串>

例:

#define sqr(x) ((x) * (x))

sqr(3+2)  ((3+2) * (3+2)) = 25

在C++中，这种替换一般被内联函数取代，进而能保证类型的正确性。（下节课内容）

例: **inline** double sqr(double x)
 {return x * x;}

宏定义的使用

3) 防止头文件被重复包含

方法一: `#ifndef`

```
#ifndef __BODYDEF_H__  
#define __BODYDEF_H__  
    // 头文件内容  
#endif
```

宏名

方法二: `#pragma once`

```
#pragma once  
// 头文件内容
```

1. 越来越多编译器支持 `#pragma once`
2. `#pragma once` 保证物理上的同一个文件不会被编译多次

宏定义的使用

```
//func.h  
int ADD(int a, int b);
```



```
//func.h  
#ifndef FUNC_H  
#define FUNC_H  
int ADD(int a, int b);  
#endif
```

增加预编译指令，防止
多次包含同一头文件时
出现编译错误

格式：

```
#ifndef 符号  
#define 符号  
内容  
#endif
```

宏定义的使用

4) 用于Debug输出等

`#ifdef` 标识符
程序段1

`#else`
程序段2

`#endif`

例:

```
// #define DEBUG
#ifdef DEBUG
    cout << "val:" << val << endl;
#endif
```

控制程序是否
输出调试信息

```
#define M(y) y*y+3*y,  
M(1+1) == 1+1*1+1+3*1+1 == 7
```

- ☒ A 7
- ☐ B 8
- ☐ C 9
- ☐ D 10

提交

C++11

- C++11标准由国际标准化组织（ISO）和国际电工委员会（IEC）旗下的C++标准委员会于2011年9月出版。C++11标准为C++编程语言的第三个官方标准，本课程内容均使用C++11标准。

1) `g++ -v` 确认g++版本 ≥ 4.7

2) 以C++11标准编译：

`g++ -std=c++11 ex6.cpp -o ex6`

3) VSCode中查看->命令窗口->Edit Configuration可编辑.vscode/c_cpp_properties.json，在c++标准选项中修改当前使用的c++标准

MAKE工具

- 使得大型编译工作自动化的一种工具
 - 减少编译程序花费的时间
 - 确保使用正确的选项进行编译
 - 确保链接正确的程序模块、程序库
- 事实上，根据MAKE的机制，还可以
 - 简化任务的重复执行过程😊
 - 减少说明文档的编写工作量 😊 😊
 - 其它创新性的想法😊 😊 😊

MAKE工具

- Makefile编写规则
 - 如果工程没有编译过，那么我们的所有cpp文件都要编译并被链接。
 - 如果工程的某几个cpp文件被修改，那么我们只编译被修改的cpp文件，并链接目标程序。
 - 如果工程的头文件被改变了，那么我们需要编译引用了这几个头文件的cpp文件，并链接目标程序。

格式: <target> : <prerequisites>
 [tab] <command>

prerequisites中如果有一个以上的文件比target文件要新的话，command所定义的命令就会被执行

百闻不如一见，来个例子吧(1)

```
# THUOOP @ 20200129
# C++ Course for THU2020
#
all: main test

main: main.cpp student.cpp
    g++ -o main main.cpp student.cpp

test: student.cpp student_test.cpp
    g++ -o test student_test.cpp student.cpp

clean:
    rm main test
```

注释以#开头

冒号为“任务”
的“条件”

冒号前为
“任务”名

指令前必须为
Tab

完成“任务”的
指令（过程）

编写 **Makefile** 的基本方法

- ✓ 不怕学习技术（值得学习与掌握的技术）
- ✓ 不怕付出劳动（绝对不会降低工作效率）

1. 从一个例子入手

2. 列出源程序清单

3. 搞清楚几个最基本的编译器参数选项

- `g++ -o`: 指定生成文件名称
- `g++ -c`: 要求只编译不链接

4. 更多Make用法 <http://www.ruanyifeng.com/blog/2015/02/make.html>






课后尝试：可用来提高效率的几个MAKE宏

- `$@` 代表目标的全名（含后缀）
- `$*` 代表无后缀的目标名
- `$<` 代表规则中的源程序名
- `%`: `[%o: %.cpp]`

运行 **Makefile** 的基本方法

- ✓ 在源代码所在目录中，打开控制台窗口，然后输入相应的命令（如下所示）

下面命令行中的  表示输入回车键

- ✓ 方法1：make 
- ✓ 方法2：make 任务名 
 - make clean 
 - make test 
- ✓ 方法3：make -f makefile的文件名 
 - make -f my_mkfile 
- ✓ 方法4：make -f makefile的文件名 任务名 
 - make -f my_mkfile test 

百闻不如一见，来个例子吧(2)

```
# Yao HaiLong @ 20200130
```

```
# C++ Course for THU2020
```

```
all: test
```

```
test: product.o sum.o main.o functions.h
```

```
    g++ product.o sum.o main.o -o test
```

```
product.o: product.cpp functions.h
```

```
    g++ -c product.cpp -o product.o
```

```
sum.o: sum.cpp functions.h
```

```
    g++ -c sum.cpp -o sum.o
```

```
main.o: main.cpp functions.h
```

```
    g++ -c main.cpp -o main.o
```

```
clean:
```

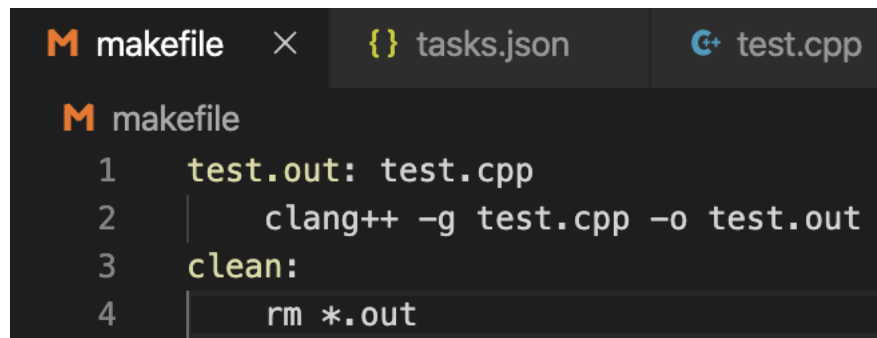
```
    rm *.o test
```

```
F:\TeachingExamples\ARGCU\Ex3>make
g++ -c product.cpp -o product.o
g++ -c sum.cpp -o sum.o
g++ -c main.cpp -o main.o
g++ product.o sum.o main.o -o test
```

IDE中使用makefile?

终端->配置任务可以打开.vscode/tasks.json配置运行相关信息

■ 编辑makefile文件



```
M makefile × {} tasks.json G+ test.cpp
M makefile
1 test.out: test.cpp
2 clang++ -g test.cpp -o test.out
3 clean:
4 rm *.out
```

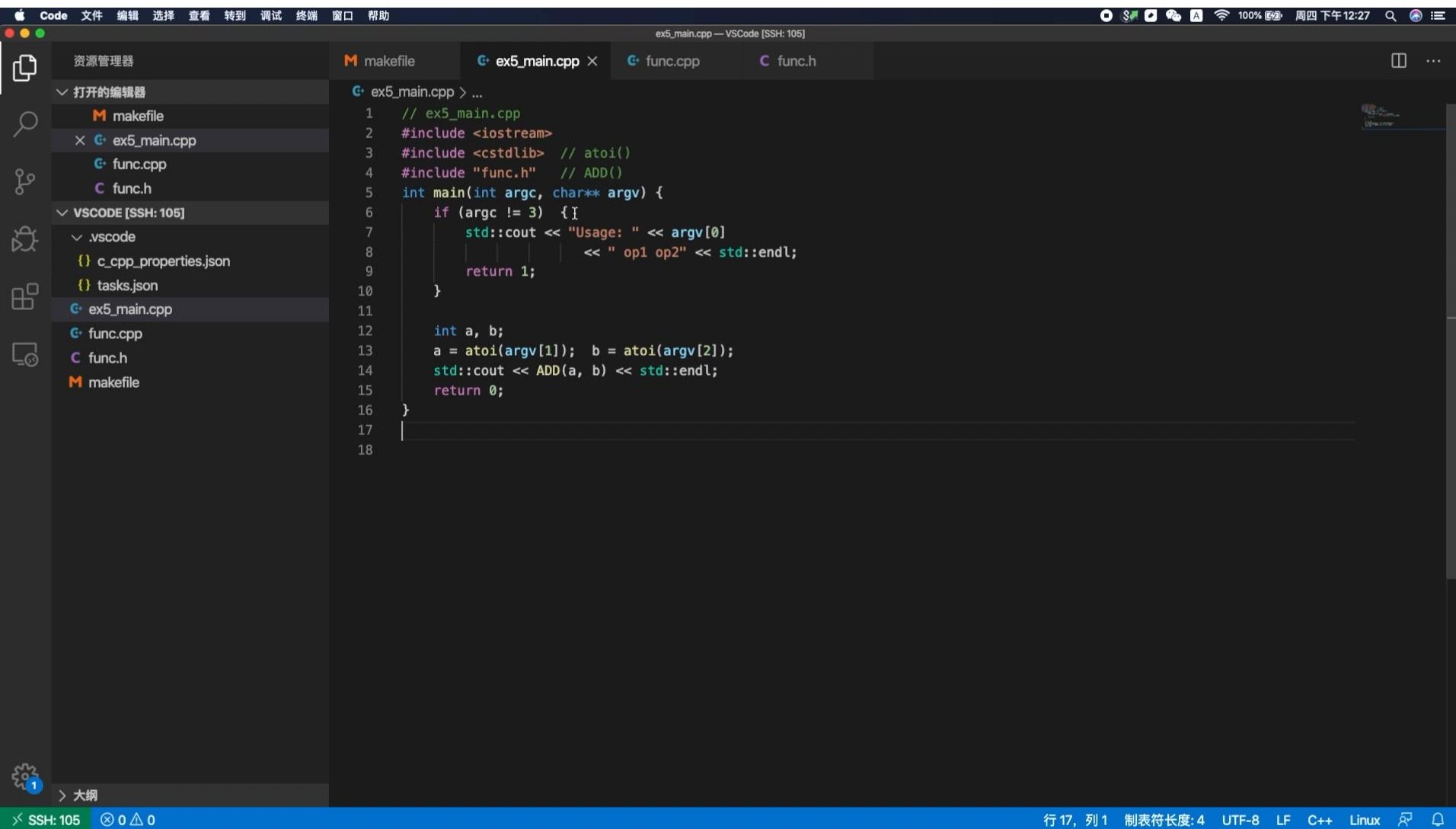
■ 编辑task.json增加运行make程序，名称为build



```
"tasks": [
  {
    "label": "build",
    "command": "make",
    "args": [],
    "type": "shell"
  },
  {
    "label": "clean",
    "command": "make",
    "args": ["clean"],
    "type": "shell"
  }
]
```

■ 终端->运行任务->选择build任务

使用makefile演示



The screenshot shows the Visual Studio Code (VS Code) editor interface. The top menu bar includes 'Code', '文件', '编辑', '选择', '查看', '转到', '调试', '终端', '窗口', and '帮助'. The status bar at the top right shows 'ex5_main.cpp — VSCode [SSH: 105]' and '周四 下午12:27'.

The left sidebar displays the '资源管理器' (Resource Explorer) with the following structure:

- 打开的编辑器 (Opened Editors)
 - makefile
 - ex5_main.cpp
 - func.cpp
 - func.h
- VS CODE [SSH: 105]
 - .vscode
 - c_cpp_properties.json
 - tasks.json
 - ex5_main.cpp
 - func.cpp
 - func.h
 - makefile

The main editor area shows the 'makefile' file with the following content:

```
1 // ex5_main.cpp
2 #include <iostream>
3 #include <cstdlib> // atoi()
4 #include "func.h" // ADD()
5 int main(int argc, char** argv) {
6     if (argc != 3) {
7         std::cout << "Usage: " << argv[0]
8         << " op1 op2" << std::endl;
9         return 1;
10    }
11
12    int a, b;
13    a = atoi(argv[1]); b = atoi(argv[2]);
14    std::cout << ADD(a, b) << std::endl;
15    return 0;
16 }
17
18
```

The bottom status bar shows 'SSH: 105', '0 0 0', and '行 17, 列 1 制表符长度: 4 UTF-8 LF C++ Linux'.

程序命令行参数

- 先看一个示例：下列程序的功能是什么？

```
// ex1.cpp @ 20200129
#include <iostream>
int main()
{
    int a, b;
    std::cin >> a >> b;
    std::cout << a + b << std::endl;
    return 0;
}
```

程序命令行参数

- EX1的特点：
 - 加法的两个操作数在程序运行时输入
 - 在被“问到”时才输入
 - 属于“强制交互”
- 能否有其他的**人机交互**方式？

main(int argc, char** argv) ?

- 请看下面的例子

```
// ex2.cpp @ 20200129
#include <iostream>
#include <cstdlib> // atoi()
int main(int argc, char** argv)
{
    int a, b;
// std::cin >> a >> b;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    std::cout << a + b << std::endl;
    return 0;
}
```

```
D:\>type ex2.cpp
// ex2.cpp @ 20090831
#include <iostream>
#include <cstdlib>      // atoi()
int main(int argc, char** argv)
{
    int a, b;
//    std::cin >> a >> b;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    std::cout << a + b << std::endl;
    return 0;
}
```

```
D:\>cl -GX ex2.cpp
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.
```

```
ex2.cpp
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

```
/out:ex2.exe
ex2.obj
```

```
D:\>ex2 4 5
9
```

```
D:\>
```

命令行参数。通过
argc, argv传入

D:\>ex2 4 5

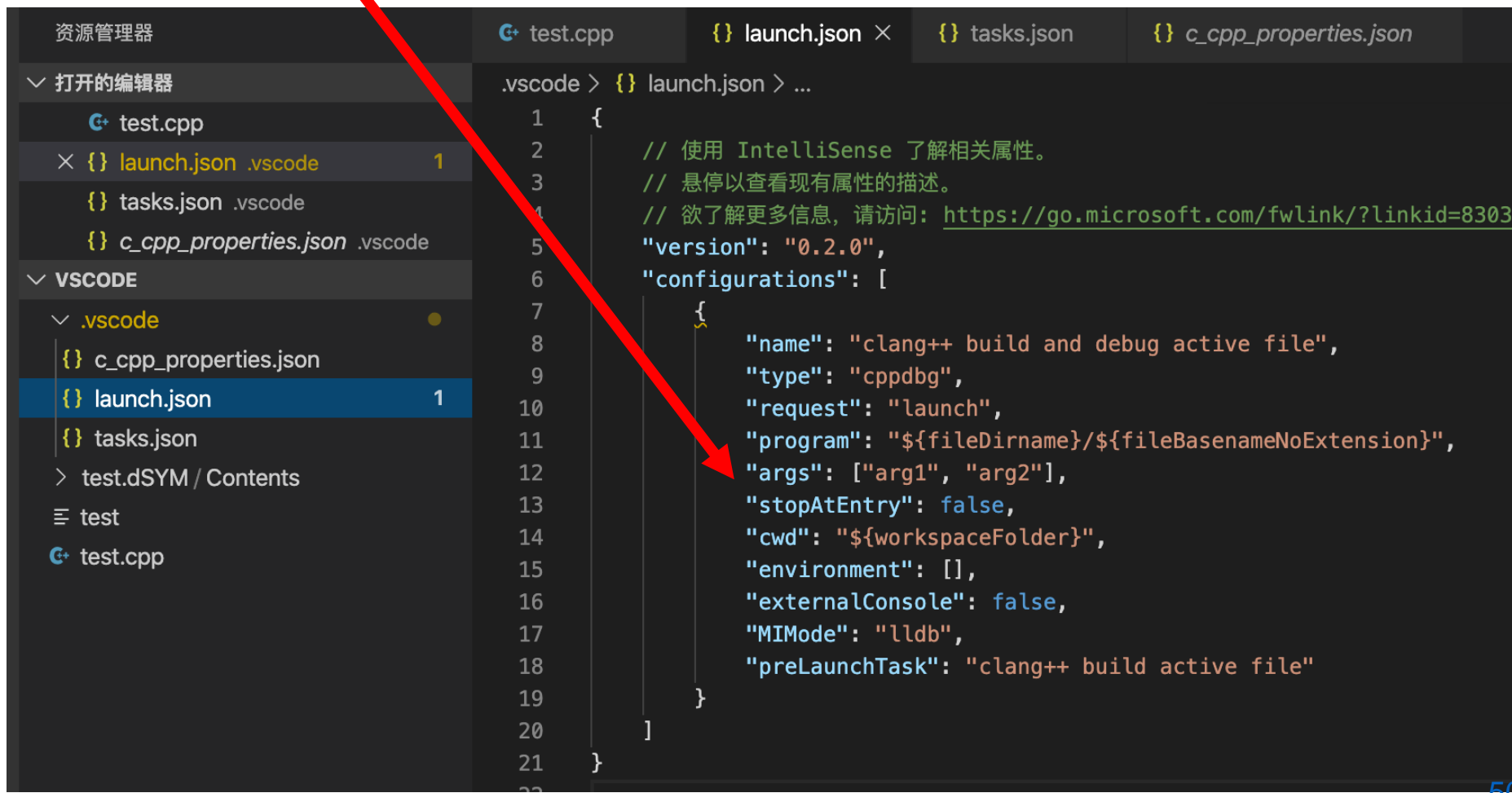
main(int argc, char** argv) ?



```
// ex2.cpp @ 20190129
#include <iostream>
#include <cstdlib> // atoi()
int main(int argc, char** argv)
{
    int a, b;
    // std::cin >> a >> b;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    std::cout << a + b << std::endl;
    return 0;
}
```

IDE中如何输入命令行参数?

查看->命令窗口->Debug open launch.json -> 修改args
(或直接修改.vscode内的配置)

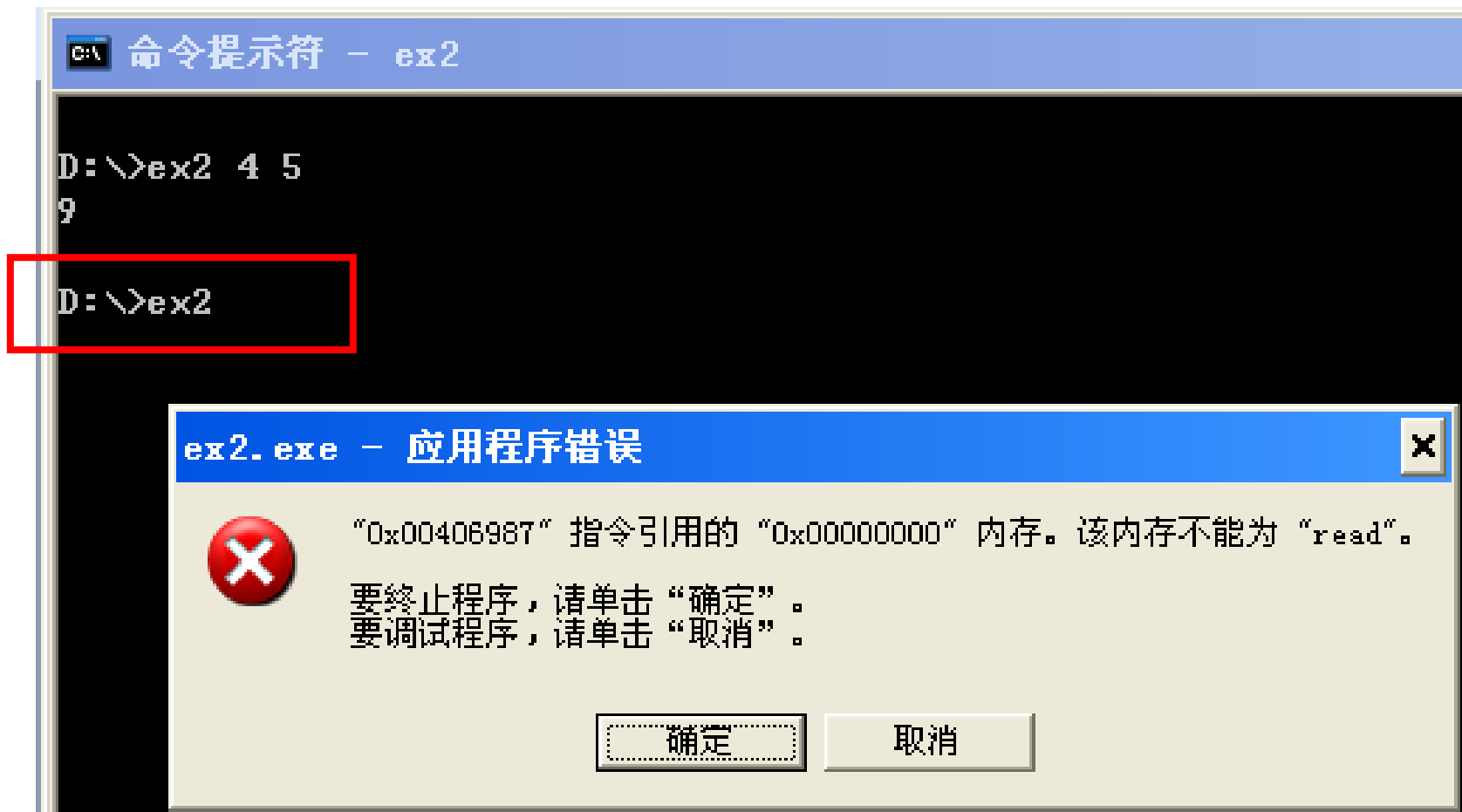


The screenshot shows the VS Code interface with the following components:

- Resource Manager (左侧):** Shows the project structure. Under the `.vscode` folder, `launch.json` is selected and highlighted in blue.
- Editor (右侧):** Displays the `launch.json` file. The file contains a JSON configuration for a debug launch. The `args` field is highlighted in blue, and a red arrow points to it from the instructions above.

```
.vscode > {} launch.json > ...
1  {
2      // 使用 IntelliSense 了解相关属性。
3      // 悬停以查看现有属性的描述。
4      // 欲了解更多信息，请访问: https://go.microsoft.com/fwlink/?linkid=8303
5      "version": "0.2.0",
6      "configurations": [
7          {
8              "name": "clang++ build and debug active file",
9              "type": "cppdbg",
10             "request": "launch",
11             "program": "${fileDirname}/${fileBasenameNoExtension}",
12             "args": ["arg1", "arg2"],
13             "stopAtEntry": false,
14             "cwd": "${workspaceFolder}",
15             "environment": [],
16             "externalConsole": false,
17             "MIMode": "lldb",
18             "preLaunchTask": "clang++ build active file"
19         }
20     ]
21 }
```

main(int argc, char** argv) ?



main(int argc, char** argv) ?

```
// ex3.cpp @ 20200129
```

```
#include <iostream>
```

```
#include <cstdlib> // atoi()
```

```
int main(int argc, char** argv)
```

```
{
```

```
    if (argc != 3) {
```

```
        std::cout << "Usage: " << argv[0]
```

```
                << " op1 op2" << std::endl;
```

```
        return 1;
```

```
    }
```

```
    int a, b;
```

```
// std::cin >> a >> b;
```

```
a = atoi(argv[1]);
```

```
b = atoi(argv[2]);
```

```
std::cout << a + b << std::endl;
```

```
return 0;
```

```
}
```

原则：总是考虑边界和异常的情况

运行EX2 4+5 =
时得到argv[1]为

☐ A EX2

☒ B 4+5

☐ C 9

☐ D =

提交

GDB调试工具

- `g++ -g a.cpp -o a.out` 编译程序
- `-g` 在可执行程序中包含标准调试信息
- `gdb a.out` 调试[a.out](#)程序
- 在gdb内不产生歧义可以简写前几个字母(红色部分)
- `run` 运行程序
- `break + 行号` 设置断点
 - `break 10 if (k==2)` 可根据具体运行条件断点
 - `delete break 1` 删除1号断点
- `watch x` 当x的值发生变化时暂停
- `continue` 跳至下一个断点

GDB调试工具

- **s**tep 单步执行(进入)
- **n**ext 单步执行(不进入)
- **p**rint x 输出变量/表达式x
 - GDB中输入 p x=1, 程序中x的值会被手动修改为1
- **d**isplay x 持续监测变量/表达式x
- **l**ist 列出程序源代码
- **q**uit 退出
- 回车 重复上一条指令

```
dmye@ubuntu:~$ cat ex3.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
int func(int s, int i) {
```

```
    return s + i;
```

```
}
```

```
int main() {
```

```
    int s = 0;
```

```
    for (int i = 1; i <= 100; ++i ) {
```

```
        s = func(s, i);
```

```
    }
```

```
    cout << s << endl;
```

```
    return 0;
```

```
}
```

```
dmye@ubuntu:~$ g++ ex3.cpp -o ex3 -g
```

```
dmye@ubuntu:~$ gdb ex3
```

求1~100的
正整数和

编译
选项

启动
调试

(gdb) b 9 在第9行设置第一个断点
Breakpoint 1 at 0x8d2: file ex3.cpp, line 9.

(gdb) r 启动程序
Breakpoint 1, main () at ex3.cpp:99

Starting program: /home/dmye/ex3

9 s = func(s, i);

(gdb) l 查看附近程序源代码

```
4                    return s + i;
5                    }
6                    int main() {
7                    int s = 0;
8                    for (int i = 1; i <= 100; ++i ) {
9                    s = func(s, i);
10                    }
11                    cout << s << endl;
12                    return 0;
13                    }
```

程序运行至
第9行暂停

```
(gdb) p s                查看变量s的值 (s==0)
$1 = 0
(gdb) n                    单步执行(不进入)
8      for (int i = 1; i <= 100; ++i ) {
(gdb) p s                查看变量s的值 (s==1)
$1 = 1
(gdb) n                    单步执行(不进入)
Breakpoint 1, main () at ex3.cpp:9
9      s = func(s, i);
(gdb) s                    单步执行(进入)
func (s=0, i=1) at ex3.cpp:4
12     return s + i;
(gdb) (回车)              重复上一条指令 step
5     }
(gdb) (回车)              重复上一条指令 step
main () at ex3.cpp:8
8      for (int i = 1; i <= 100; ++i ) {
```

```

(gdb) p s          查看变量s的值 (s==3) 执行了两次更新
$1 = 3
(gdb) n            单步执行 (不进入)
9                 s = func(s, i);
(gdb) c            跳至下一个断点 (进行了一次循环)
Continuing
Breakpoint 1, main () at ex3.cpp:9
Breakpoint 1, main
() at ex3.cpp:9
9                 s = func(s, i);
(gdb) p s          查看变量s的值 (s==6) 执行了三次更新
$1 = 6
(gdb) b 11         在第11行设置第二个断点
Breakpoint 2 at 0x5555558..: file ex3.cpp, line 11.
(gdb) d break 1    删除第一个断点
(gdb) info b       查看断点信息
Num   Type          Disp  Enb  Address      What
2     breakpoint    keep   y    0x0000..    in main() at..

```

(gdb) c 跳跃至下一断点 (跳出循环)

Continuing.

Breakpoint 2, main () at ex3.cpp:11

11 cout << s << endl;

(gdb) p s 查看变量s的值 (s==5050)

\$1 = 5050

(gdb) q 退出程序

A debugging session is active.

Inferior 1 [process 22223] will be killed.Quit
anyway? (y or n) y

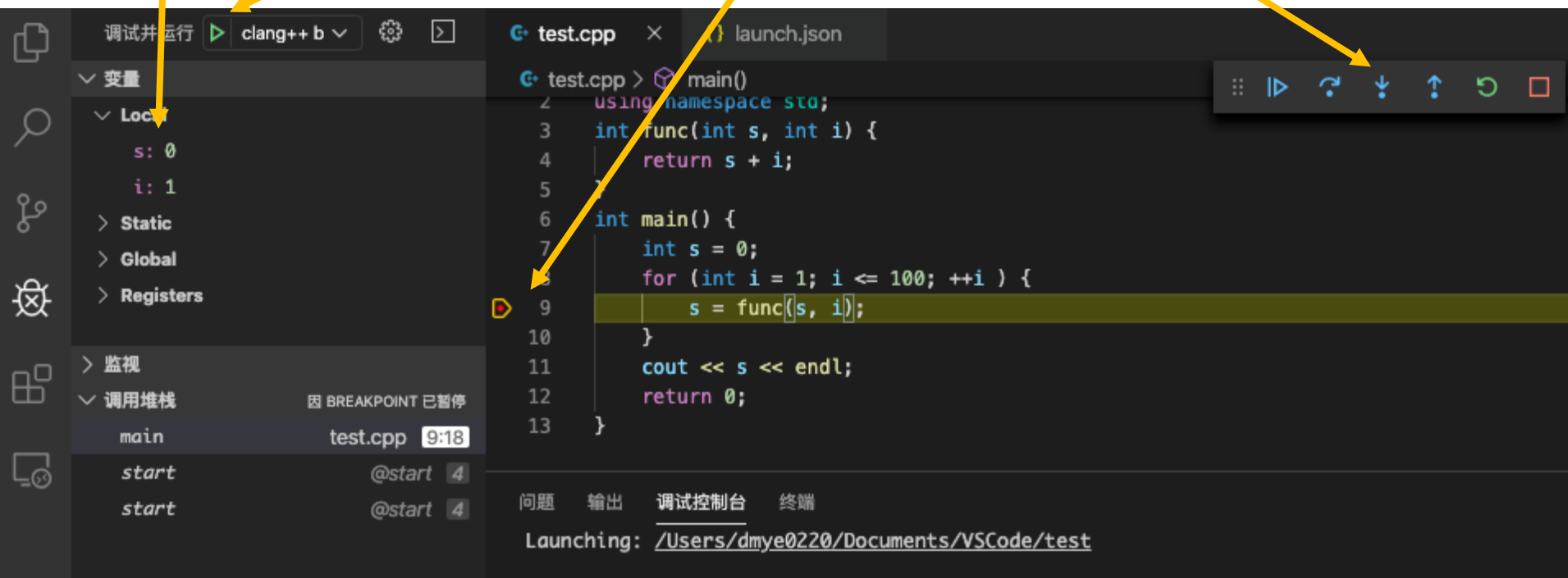
dmye@ubuntu:~\$

VSCode调试

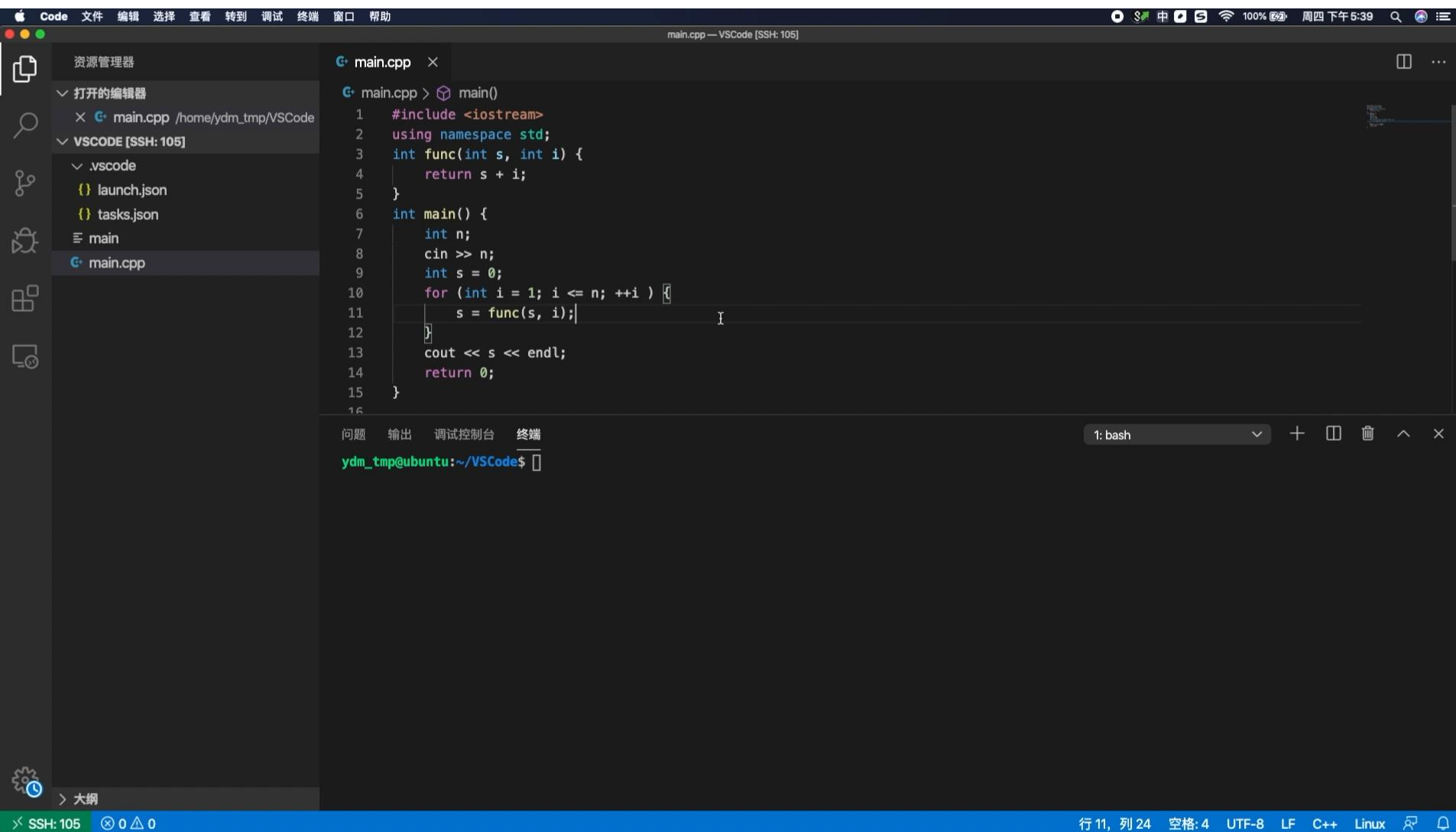
③查看变量值 ②启动调试(F5)

①设置断点

④单步执行



调试演示



灵活搭配调试方法 (举例)

1

```
// ex4.cpp @ 20200129
#include <iostream>
int main() {
    int a[4], b[4];
    for (int i = 0; i <= 4; i++) {
        a[i] = i;
    }
    std::cout << b[0] << std::endl;
    return 0;
}
```

此时使用 `watch b[0]` 可在 `a[4]=4` 之后停下来发现访问内存越界的问题，`b[0]` 的值被修改为了4

2

当程序使用 `-O2` 和 `-O3` 优化后，程序运行结果不同或者报错，不能使用 `gdb` 调试。此时可使用输出调试法，在程序各个位置设置 `print` 语句输出中间值。

结束

课后阅读：

- 《C++编程思想》 3.11 make: 管理分段编译
- 关于make的高级用法

<http://www.ruanyifeng.com/blog/2015/02/make.html>

基本尝试

- 编写一个小程序，体会多个文件的编译和链接（多个h、cpp文件）。设计一些全局函数和变量，在多个cpp文件中引用它们。
- 编写一个makefile文件，对这些文件实现自动关联的编译

高级尝试 -- 理解一个开源项目中的Makefile

<https://github.com/moses-smt/giza-pp/>

.PHONY: gizapp mkcls-v2

all: gizapp mkcls-v2

gizapp:

\$(MAKE) -C GIZA++-v2

mkcls-v2:

\$(MAKE) -C mkcls-v2

clean:

\$(MAKE) -C GIZA++-v2 clean

\$(MAKE) -C mkcls-v2 clean

📁 GIZA++-v2

📁 mkcls-v2

📄 Makefile

📄 README

.PHONY后内容无视目标文件存在与否都执行command, 避免与工作目录下同名文件夹冲突

定义变量宏:

Var = 字符串

调用变量 \$(Var)

预定义: \$(MAKE) = make

make -C [target]:

切换到[target]文件夹执行make命令

高级尝试 -- 理解一个开源项目中的Makefile

子文件夹GIZA++-v2中Makefile文件:

Part1:

.SUFFIXES: .out .o .c .e .r .f .y .l .s .p .cpp .alpha2o .pentiumo .sgio .alphao # 该Makefile所支持后缀类型

INSTALLDIR ?= /usr/local/bin/ # 使用“?=”进行赋值的时候如果该变量已经赋值过了，那么将跳过

CXX = g++ # 后面统一使用 CXX 编译

CFLAGS = \$(CFLAGS_GLOBAL) -Wall -Wno-parentheses
CFLAGS_OPT = \$(CFLAGS) -O3 -funroll-loops -DNDEBUG -
DWORDINDEX_WITH_4_BYTE -DBINARY_SEARCH_FOR_TTABLE
CFLAGS_PRF = \$(CFLAGS) -O2 -pg -DNDEBUG -
DWORDINDEX_WITH_4_BYTE
... ..
LDFLAGS =

高级尝试 -- 理解一个开源项目中的Makefile

子文件夹GIZA++-v2中Makefile文件:

Part2:

```
// Makefile.src
include Makefile.src      SRC = Parameter.cpp myassert.cpp ...

OBJ_DIR_OPT = optimized/
OBJ_OPT = ${SRC:%.cpp=$(OBJ_DIR_OPT)%.o} # %为匹配符
          # 提取SRC中所有.cpp的前缀, 构成OBJ_DIR_OPT+前缀值.o的集合
OBJ_DIR =

GIZA++: $(OBJ_DIR_OPT) $(OBJ_OPT) # 将.o文件一起链接为GIZA++
          $(CXX) $(OBJ_OPT) $(LDFLAGS) -o GIZA++

$(OBJ_DIR_OPT): $(OBJ_DIR)          # 创建文件夹
    -mkdir $(OBJ_DIR_OPT)

$(OBJ_DIR_OPT)%.o: %.cpp # 编译所有cpp
    $(CXX) $(CFLAGS_OPT) -c $< -o $@
```

\$@ --目标文件
\$^ --所有的依赖文件
\$< --第一个依赖文件